Fault-Tolerant Synthesis using Non-Uniform Redundancy

Keven L. Woo, Matthew R. Guthaus
Department of CE, University of California Santa Cruz, Santa Cruz, CA 95064
{kwoo,mrg}@soe.ucsc.edu

***Abstract*— As process technologies continue to scale into the nanometer regime, devices are becoming significantly more unreliable. Many forms of unreliability manifest as transient faults and can cause intermittent random logic upsets. These logic upsets are often caused by natural radiation (neutrons and alpha particles) or on-chip noise (cross-coupling, supply drop, or flicker noise). This research improves reliability by using non-uniform redundancy. Specifically, we present a dynamic programming algorithm that considers many possible topological redundancies, yet maintains a linear run-time due to efficient pruning of suboptimal solutions. Our algorithm provides designers with a Pareto-optimal set of solutions that trade reliability for area. Compared to existing Triple Modular Redundancy (TMR), we see similar reliability with only 35% area overhead instead of 326%.**

I. Introduction

One of the biggest challenges in continued device scaling is system reliability. As individual devices and voltages shrink, sources of noise such as radiation (neutrons and alpha particles) become more significant. These sources of noise, however, are transient and only occur sporadically.

Computing pioneer von Neumann developed the preeminent theory in the area of fault-tolerant synthesis back in the 1950's when he proposed to build reliable computers out of unreliable components (vacuum tubes) [1]. He laid the theoretical framework, then called NAND multiplexing, for a vast array of works that proposed to use redundancy to create reliable devices. His theory is well developed, but the applicability to real circuits is troublesome. Von Neumann's original proposal required redundancy of many thousand times in order to obtain reliable computers. Obviously, this is not practical because of the significant increase in area.

Many practical works have focused on system or block-level design and have proposed Triple Modular Redundancy (TMR) [1], N-tuple Modular Redundancy (NMR) [2], Cascaded TMR (CTMR) [3], and Recursive Triple Modular Redundancy (RTMR) [4]. Other recent works examined the granularity (e.g. gate-level and module-level) at which redundancy should be applied [5].

Several works have examined input-dependent fault analysis [6]–[8], but they have not proposed algorithms for redundancy insertion that considers input-dependence. Some works in the area of radiation-tolerant circuit design have considered input-dependence. These works included sizing a logic gate [9] and adding shadow logic to detect faults [10], but they did not consider non-uniform redundancy.

This paper proposes two enhancements to improve the reliability of circuits. First, redundancy can occur at many granularities. These granularities should not be restricted to the gate or module-level. Thus, any sub-circuit could be made redundant. Second, redundancy can be non-uniform to account for input-dependent behavior and variations in device vulnerability.

Our paper is organized as follows: Section II provides an overview of our approach; Section III describes our core dynamic programming algorithm; Section IV describes the corresponding design and experimental methodology; Section V presents our experimental results; and Section VI makes our conclusions.

II. Approach

Our methodology produces reliable designs in unreliable technologies by adding non-uniform redundancy. The goal of our approach is to find a better use of redundancy than the traditional TMR, NMR, or RTMR methods. From a cost perspective, if a large sub-circuit is made redundant, the duplication cost of TMR is high, but the cost of the majority gate is amortized. However, if the sub-circuit is small, the duplication cost is low, but the relative cost of the majority gate is large. In addition, a majority gate itself can be susceptible to transient errors. We leverage that circuits have non-uniform vulnerability and non-uniform observability when selecting sub-circuits to make redundant.

For our fault model, we assume that the probability of a fault is an independent, constant value per unit of area since gate area is approximately equal to the risk exposure of a gate to radiation sources. A larger gate will have more surface area and, therefore, is more likely to be struck with a neutron or an alpha particle. Unless otherwise mentioned, the probability of a fault is 1×10^{-4} per μm^2. This means that a small (X1) inverter with an area of $0.532 \mu m^2$ has a probability of 5.32×10^{-5} of having a faulty output; this is the vulnerability. We study the impact of this vulnerability number in our later studies. Sub-circuits with a high vulnerability are good candidates for redundancy.

A faulty value, however, is not always visible at the outputs of the circuit due to logical masking in the circuit. In addition to the probability of an error occurring (vulnerability), circuit reliability is also dependent on the error propagating through a circuit to the outputs (observability). The observability of a fault can be computed using Monte Carlo fault simulations or static methods [6]–[8]. The observability of a fault depends on the logic switching characteristics of a given circuit. In general, gates that are closer to the outputs

978-1-4244-5028-2/09/$25.00 ©2009 IEEE

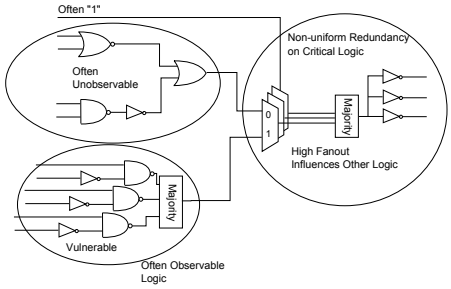


Fig. 1. Example of non-uniform redundancy.

and gates with large fan-out cones are more observable. Sub-circuits with highly observable outputs are good candidates for redundancy.

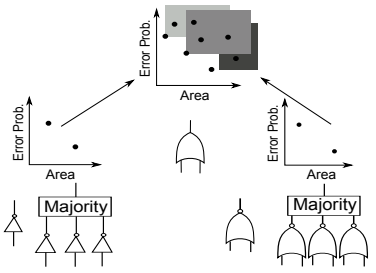
The capabilities of our algorithm are best exemplified by Figure 1, where a mux typically selects a single input due to a frequent select input of 1. The frequent operating behavior renders half of the logic virtually unobservable. If one portion of a circuit is extremely vulnerable, observable, or both, that portion should be made redundant. This is exemplified by the high-fanout multiplexor, which has a large fanout and is therefore more observable than the other logic. For the fan-in cone of a multiplexor, it is sufficient to make only half of the fan-in logic redundant due to the rare observability of the other logic. This can result in significant area savings with almost no impact on reliability.

Given these observations on vulnerability and observability, we can conclude that in order for a gate to have a reliable output, its inputs must also be reliable. In addition, we cannot judge the reliability of a gate until we know the observability and reliability of its fan-out cone and the other fan-ins to the outputs. Together, these characteristics make an ideal argument for a dynamic programming algorithm which we will now present.

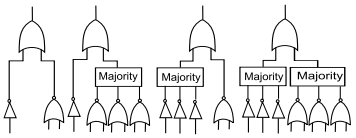
III. Non-Uniform Redundancy using Dynamic Programming

Our core algorithm optimizes a single output and its fan-in logic cone. In our approach, the logic circuit is decomposed into a set of trees with each primary output as a tree root similar to traditional technology mapping [11]. In Section IV, we explicitly describe our decomposition methodology for general multi-output circuits.

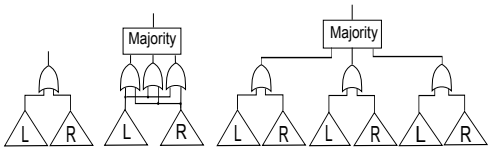
Our algorithm performs a bottom-up enumeration of all possible redundancies at every topological level. Figure 2(a) shows a simple example of how our dynamic programming algorithm would calculate the redundancy for a three gate circuit (a NOR and an INV feeding into an OR gate). The dynamic programming traverses the tree in a bottom-up manner constructing potential redundancy implementations of the sub-trees. This starts by making NOR and INV solutions with and without redundancy. A single majority gate is added in the case of redundancy. Each of these solutions has a different area versus error probability trade-off as shown by each corresponding plot.



(a) Example of bottom-up solutions and pruning for redundancy.



(b) Example of all the possible combinations of the child sub-trees enumerated with an OR gate



(c) Example of the same circuit with no redundancy (left), SGR (middle) and NUTMR (right).

Fig. 2. Dynamic programming implementation of selective redundancy.

At each higher node in the tree, such as the OR gate, all possible combinations of the child sub-trees enumerated as shown in Figure 2(b). In our example, there would be four candidate combinations since each sub-tree has two solutions.

For each of the candidate combinations, there are three ways to combine them in our approach. Figure 2(c) explicitly shows the three methods to combine two child sub-trees, L and R, with a root gate:

1) The simplest way is to simply add the root gate to the children candidates with no additional redundancy. Note that the L and R sub-trees may already contain redundancy at previous levels in the trees.
2) Another way is to make only the root gate, the OR, redundant and add a majority gate. No additional redundancy is added to the L and R sub-tree candidates, however. We term this single gate redundancy (SGR).
3) The last alternative is to attach a single copy of the root gate, but then perform traditional TMR on the entire resulting solution. We call this non-uniform TMR (NUTMR).

In addition to no redundancy, SGR and NUTMR, it is possible to implement redundancy candidates of varying depths between SGR and NUTMR or to add different amounts of redundancy (NMR), but we leave this for future work. Two extreme final solutions are shown in Figure 3 for brevity: one with no redundancy and one with NUTMR at all possible levels.

The overview of the dynamic programming function is

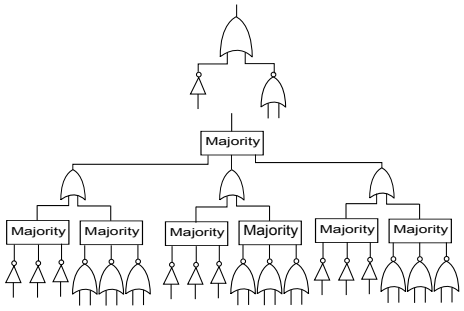


Fig. 3. Two extreme solutions of no redundancy (top) or non-uniform TMR at all levels (bottom).

shown in Algorithm 1. It first sorts the gates in the logic cone by topological order and processes each gate from the inputs to the output. At each level, it enumerates the combinations of the left and right candidate sub-solutions and adds solutions for the current level with no redundancy, SGR, and NUTMR using the functions ADD_NO_MR, ADD_SGR, and ADD_TMR, respectively. Once this is done, the resulting candidate sub-solutions are pruned.

Algorithm 1 DP_OPT(Cone)

```
Input: A Cone
Output: A set of Pareto-optimal solutions for that cone

SORT(Cone) // by bottom-up topological order
for each gate i ∈ Cone do
  for each candidate l ∈ i.left.Candidates do
    for each candidate r ∈ i.right.Candidates do
      i.Candidates ← {i.Candidates,ADD_NO_MR(i,l,r)}
      i.Candidates ← {i.Candidates,ADD_SGR(i,l,r)}
      i.Candidates ← {i.Candidates,ADD_TMR(i,l,r)}
    end for
  end for
  i.Candidates ← PRUNE(i.Candidates)
end for
```

We then prune many sub-optimal solutions after they are generated during the bottom-up enumeration. Our dynamic programming algorithm uses probabilistic error analysis to generate an optimal trade-off curve of area versus the probability of a circuit error, P_e. The observability will be a constant for any possible equivalent solution so we can safely ignore this part of the reliability measurement.

For the time being, we compute P_e by running Monte Carlo fault simulations on the sub-optimal candidates to compute the observability. More specifically, input values are randomly selected and the circuit is then simulated with and without random faults. When a fault occurs in a gate, a logical flip of the output is performed (e.g. 0→1 or 1→0). If any of the outputs of the faulty simulation do not match the correct simulation, it is considered an error. P_e is simply the number of simulations with any errors divided by the number of iterations. For speed improvement, 64-bit (a long int) logical values are simulated in parallel. Any static method such as [7] can reduce run-time. It is extremely important that the fault analysis include switching correlation, however, since each instance of internal redundancy adds reconvergent fanout which introduces correlated behavior. Ignoring this correlation results in pessimistic P_e values and keeps incorrect candidate solutions.

Since we do not know the reliability of other sibling subtree solutions, we do not know how reliable to make our current solution. If the sibling solutions are very unreliable, we may waste area by making our current solution too reliable. If the sibling solutions are very reliable, we may restrict the overall reliability of the circuit due to the unreliability of the current sub-solution. Therefore, we keep all co-optimal candidate solutions and postpone the decision of the best candidates until the end of the bottom-up traversal.

Pruning the sub-optimal solutions is equivalent to finding the co-optimal candidates. To find the co-optimal candidates, we compare all pairs of solutions, A and B, and, if $P_e(A) \geq P_e(B)$ and $Area(A) \geq Area(B)$, solution A is considered sub-optimal since it can easily be replaced by solution B in all situations. This is shown in Figure 2(a) where three solutions in the merged Pareto curve dominate all of the other solutions in the gray areas. All the solutions in the gray areas are pruned since they are unnecessary. Pruning limits the number of solutions to a Pareto-optimal set and a roughly linear number of items, thus, providing a more efficient algorithm. The pruning algorithm itself, as shown in Algorithm 2, is an $O(n \log n)$ algorithm. The candidates are first sorted by increasing area and then a single pass over the sorted array finds the non-dominated solutions in linear time.

Algorithm 2 PRUNE(Candidates)

```
Input: Candidates is an array of candidate implementations
Output: Solutions is an array of non-dominated candidates

SORT(Candidates) // by increasing area then P_e
i ← 1
for j ← 2 to length[Candidates] do
  if AREA(Candidates[i]) < AREA(Candidates[j]) and
  P_e(Candidates[i]) > P_e(Candidates[j]) then
    Solutions ← {Solutions, Candidates[i]}
    i ← j
  end if
end for
Solutions ← {Solutions, Candidates[i]}
```

IV. Methodology

Our methodology for multiple output circuits involves three steps: dividing the circuit into logic cones, optimizing each logic cone sequentially, and combining the logic cones.

A. Finding Cones

Since our dynamic programming algorithm can only deal with single output circuits, we must first decompose multi-output circuits into a set of single output logic cones. To do

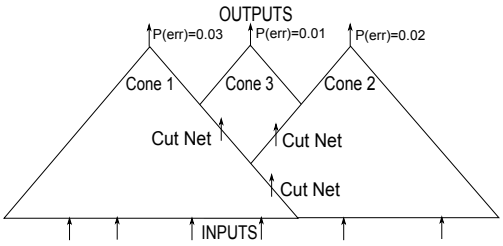


Fig. 4. Example of cone decomposition, cut nets, and topological cone ordering.

this, we perform an initial fault analysis and rank the outputs in order of increasing probability of error, P_e. Figure 4 shows an example of a general circuit that is decomposed into three cones.

For each output, we traverse the transitive fan-in logic until we reach a primary input or we reach a gate that has already been claimed by another logic cone. When a gate is visited, it is marked as belonging to a logic cone. After doing this for each output, we are left with a collection of non-overlapping cones. It is possible that cones have reconvergent fanout.

When splitting the logic cones, we mark any nets that fan out from one cone to another as a "cut net" as shown in Figure 4. These cut nets are treated as primary inputs of the destination logic cone. They are not treated as a primary outputs of the source logic cone, however, so they are not directly optimized.

B. Cone Optimization

Once we have the logic gates divided into cones, we optimize the cones in the order of the increasing output error probability, P_e, using our dynamic programming algorithm in Section III. This effectively optimizes the cones in bottom-up topological order.

Whenever we encounter a cut net, the cone that drives the net is already optimized so we select the worst P_e over all candidate implementations and use this as the input to the downstream logic cone. We do this, because we do not know which implementation of the cone will be used until all cones are solved. This provides a conservative estimate of P_e on that input during the optimization of the cone. All primary inputs are assumed to be error free, but this restriction can be easily relaxed.

C. Combining the Cones

When all of the cones have been optimized, we are left with a small number of Pareto-optimal implementations of each cone (typically, 3-10). We then create the final solutions of the entire circuit by picking an implementation of each cone and pruning sub-optimal solutions as before.

Specifically, the pseudo-code of our algorithm is shown in Algorithm 3. We start with an initial set of sub-solutions by adding all the implementations of the first cone. The algorithm then iteratively combines these sub-solutions with the implementations of the next cone. The MERGE function does this by enumerating all combinations of current sub-solutions (Final) with each implementation j of $Cone[i]$.

Algorithm 3 COMBINE(Cone)

Input: Cone is a 2D array of (cones $\times$ implementations)
Output: Final is an array of solutions of whole circuit

```
Final ← Cone[1]
for i ← 2 to length[Cone] do
  for each implementation j ∈ Cone[i] do
    Final ← MERGE(Final, j)
  end for
  Final ← PRUNE(Final)
end for
```

These new sub-solutions are collected and replace the last set of sub-solutions. Even though the number of implementations of each cone is small, we must prune the final sub-solutions before enumerating the combinations with the next cone to prevent exponential run-time. In general, our pruning algorithm removes up to 90% of the candidates during the combination of each cone.

V. Experimental Results

We implemented our tool in C++ using OpenAccess 2.2.6 and ran our experiments on AMD dual-core Operton 2218 processors with 8GB of memory. For our fault analysis, two million iterations of Monte Carlo were used.

For our studies, we synthesized the ISCAS benchmarks [12] using Synopsys Design Compiler and the Nangate 45*nm* Open Cell library [13]. The synthesis target was for minimum area given an input to output operating frequency of 200Mhz. The timing constraints assume all the inputs come from flip-flops and outputs are captured by flip-flops. The clock has 100ps skew.

Since the Nangate library does not include a majority gate, we created one by using 40% of the area of a Full-Adder. This corresponds to the approximate area percentage that the majority portion of a mirror adder cell occupies.

A. Basic Results

In Table I, we show a comparison between the original, a traditional TMR circuit, and our most reliable circuit, which is a combination of our SGR and NUTMR methods. The TMR circuit is the original circuit replicated three times with a majority gate added to each of the outputs. The area and P_e are normalized to the original circuit without any redundancy. In most of the benchmarks, our results are comparable to a TMR circuit in terms of reliability. However, our results typically have 35% more area compared to the original design whereas the TMR circuit uses 326% more area on average. On average, our reliability is very similar to TMR as shown by the 47% and 51% of the original error rate. In some cases, our circuit attains better reliability (e.g. c880, c2670) and in others it is slightly worse (e.g. c17, c432, c1908). It should be noted that TMR actually makes c2670 more unreliable than the original circuit because c2670 has many outputs and the many added majority gates decrease the overall reliability.

TABLE I

COMPARISON OF ORIGINAL CIRCUIT, TRADITIONAL TMR AND OUR ALGORITHM. AREA AND ERROR PROBABILITY ARE NORMALIZED TO ORIGINAL CIRCUIT.

	Original			TMR		Ours	
Bench.	Area (μm^2)	Area	P_e	Area	P_e	Area	P_e
c17	5.6	1.0	1.0	3.68	0.86	1.63	0.92
c432	111.2	1.0	1.0	3.12	0.34	1.33	0.51
c499	229.6	1.0	1.0	3.27	0.52	1.72	0.52
c880	226.4	1.0	1.0	3.22	0.41	1.33	0.34
c1355	231.4	1.0	1.0	3.26	0.52	1.71	0.53
c1908	238.3	1.0	1.0	3.20	0.33	1.41	0.47
c2670	424.0	1.0	1.0	3.63	1.08	1.19	0.61
c3540	652.0	1.0	1.0	3.06	0.17	1.07	0.39
c5315	916.6	1.0	1.0	3.23	0.48	1.29	0.41
c7552	1122.3	1.0	1.0	3.18	0.37	1.15	0.48
c6288	1672.6	1.0	1.0	3.04	0.04	1.06	0.43
Avg.	530.0	1.0	1.0	3.26	0.47	1.35	0.51

We obtained more area savings compared to a TMR circuit because our algorithm implements redundancy on the most vulnerable and observable areas of the circuit. Upon examining the final solutions, it is evident that high-fanout sub-trees and gates closer to the outputs are made redundant.

In Figure 6, the line labeled "SGR" and "NUTMR" presents an example of a Pareto curve of benchmark c880. The Pareto curves of the other benchmarks show similar trends. Most of the circuits achieve the best reliability within a 30-60% increase in area. Beyond this, no improvement in reliability is possible given our limited set of single-gate and TMR redundancy operations during the dynamic programming.

B. Error Rates

Exact error rates are often difficult to quantify. Radiation, for example, depends on the elevation, solar flare activity, nearby radioactive elements, etc. The error rate also has a significant impact on, not only the reliability, but the structure of non-uniformly redundant solutions.

Figure 5 shows a comparison of the Pareto curves for different error rates on benchmark c432. The Pareto curves show our method is more effective with higher error rates. This is because the relative error rate of the majority gate compared to a sub-tree is less significant. Therefore, adding redundancy has more opportunity for improvement. With a lower error rate, it takes considerably more redundancy to make improvements to a circuit's reliability. With an error rate of 1×10^{-4} per μm^2, there is only a small improvement with a 50% increase in area. In contrast, there is nearly a $2\times$ improvement in reliability for the same area increase when the error rate is 1×10^{-2} per μm^2. This means that for highly unreliable technologies, this methodology will show better results.

C. Single-Gate Redundancy (SGR) vs. Non-Uniform TMR (NUTMR)

In Section III, we implemented two different redundancy schemes: SGR and NUTMR. In this section, we study the effectiveness of each scheme independently. Figure 6 is a graph of benchmark c880 which shows three Pareto curves:

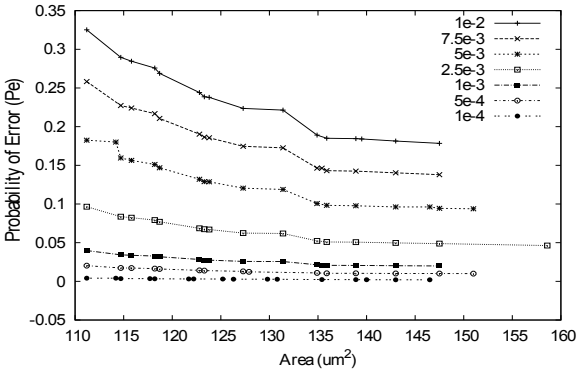


Fig. 5. Pareto curve for c432 with different error rates.

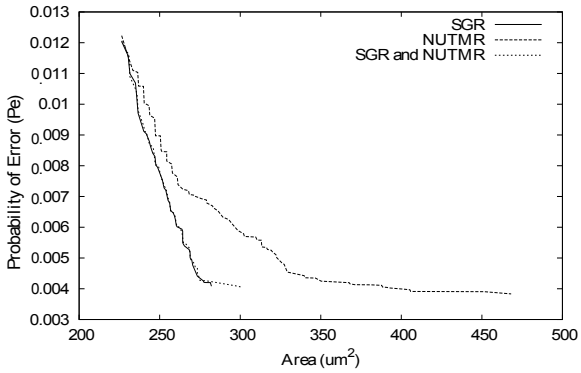


Fig. 6. A graph of c880 comparing SGR, NUTMR and both.

using only SGR, NUTMR and both SGR and NUTMR simultaneously. From the SGR and NUTMR curves, SGR tends to create circuits that use less area than NUTMR circuits. On the other hand, NUTMR redundancy has a slightly better probability of error in the extreme.

Table II shows a comparison between the original circuit and our most reliable circuit for SGR, NUTMR and both SGR and NUTMR simultaneously. In all of benchmarks, the SGR circuit has a much smaller area than the NUTMR circuit. There is from 40 to 50% less area except in benchmark c1355. The SGR solution mainly made gates that are closer to each output redundant, while NUTMR implemented redundancy on internal high-fanout gates.

In addition, c1355 had very few high-fanout gates and most of these were near the inputs. In most of the benchmarks, except c1355, the best NUTMR circuit has a better reliability than the best SGR circuit.

We also studied this comparison with higher error rates. SGR and NUTMR showed similar trends in that our best NUTMR circuits had better reliability, while our best SGR circuits used much less area. Using a combination of these methods may lead to improved results because SGR leans towards using smaller area, while NUTMR tends to lower *Pe*.

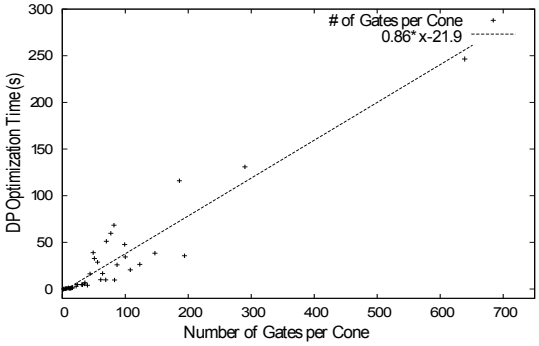
D. Run Time

The actual complexity of our algorithm depends on the topology. This makes it difficult to formally quantify the ef-

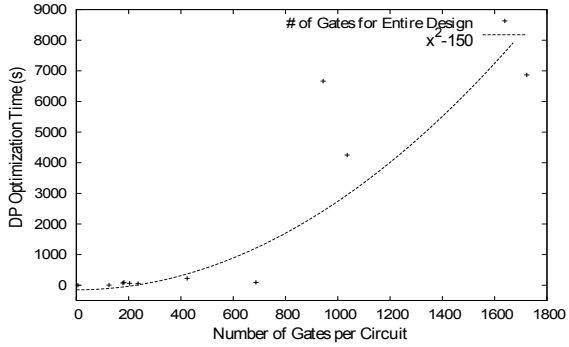
TABLE II

COMPARISON OF ORIGINAL CIRCUIT, SGR ONLY, NUTMR ONLY AND BOTH SGR AND NUTMR SIMULTANEOUSLY. AREA AND ERROR PROBABILITY ARE NORMALIZED TO ORIGINAL CIRCUIT.

	Orig.		SGR		NUTMR		Both	
Bench.	Area	P_e	Area	P_e	Area	P_e	Area	P_e
c17	1.0	1.0	1.63	0.96	2.77	0.91	1.63	0.92
c432	1.0	1.0	1.33	0.51	2.89	0.36	1.33	0.51
c499	1.0	1.0	1.71	0.53	3.31	0.53	1.72	0.52
c880	1.0	1.0	1.24	0.34	2.07	0.31	1.33	0.34
c1355	1.0	1.0	1.71	0.53	1.74	0.59	1.71	0.53
c1908	1.0	1.0	1.37	0.49	3.21	0.35	1.41	0.47
c2670	1.0	1.0	1.21	0.61	2.90	0.61	1.19	0.61
c3540	1.0	1.0	1.09	0.37	2.72	0.34	1.07	0.39
c5315	1.0	1.0	1.28	0.43	2.04	0.64	1.29	0.41
c7552	1.0	1.0	1.13	0.51	1.57	0.59	1.15	0.48
c6288	1.0	1.0	1.04	0.54	2.97	0.34	1.06	0.43
Avg.	1.0	1.0	1.34	0.53	2.56	0.51	1.35	0.51



(a) Empirical run-time for dynamic programming.



(b) Empirical run-time for overall algorithm.

Fig. 7. Empirical run-times of our dynamic and overall algorithm.

fect of pruning during our algorithm. Figure 7(a) shows a plot of the number of gates and the amount of time to optimize a single cone using our dynamic programming algorithm in Section III for a variety of cones in the benchmarks. This shows empirically that our run-time is linear with the number of gates per cone. The run-time shown does not include the time spent in Monte Carlo fault analysis since we plan to replace that with a static fault analyzer.

Figure 7(b) shows the run-time of the overall algorithm for all of the benchmarks. It is weakly quadratic due to the enumeration during the combining of the cones. A better algorithm will be presented in the future.

VI. CONCLUSION AND FUTURE WORK

In conclusion, we presented a dynamic programming algorithm and a framework that provides an alternative to traditional redundancy methods. We offer similar reliabilities to TMR with area overheads of only 35% compared to the original circuit. In addition, we provide the designer with a Pareto-optimal set of solutions which allows them to choose the best solution for the application at hand.

Using our infrastructure, we showed that single-gate redundancy, to our surprise, offers better reliability for a given area than traditional TMR during our dynamic programming algorithm. In the future, we plan to investigate other redundancy methods such as k-level redundancy in the same framework. In this method, we could perform TMR up to k-levels deep in the subcircuit which will offer another dimension to our algorithm. In addition, we can also easily incorporate n-tuple modular redundancy (NMR), cascaded TMR (CTMR) and Recursive TMR (RTMR) into our algorithm. Arbitrarily adding more variants to our algorithm may or may not provide significantly improved results. This is to be determined.

In the future, we plan to consider performance in addition to reliability and area. Currently, we do not buffer signals and do not consider the effect of fanout on our circuit's performance. We also plan to integrate a static fault analyzer rather than use Monte Carlo. In addition, we plan to compare our results to other redundant methods such as NMR, CTMR and RTMR. These tasks are left as future work.

REFERENCES

[1] J. von Neumann, "Probabilistic logic and the synthesis of reliable organisms from unreliable components," *Automata Studies*, pp. 43–98, 1956.

[2] F. P. Mathur and A. Avizienis, "Reliability analysis and architecture of a hybrid redundant system: Generalized triple module redundancy with self-repair," in *AFIPS*, vol. 36, 1970, pp. 375–383.

[3] K. Nikolic, A. Sadek, and M. Forshaw, "Architectures for reliable computing with unreliable nanodevices," in *Nanotech*, 2001, pp. 254–259.

[4] D. D. Thaker, R. Amirtharajah, F. Impens, I. L. Chuang, and F. T. Chong, "Recursive TMR: Scaling fault tolerance in the nanoscale era," *IEEE Design and Test*, pp. 298–305, 2005.

[5] D. Bhaduri and S. K. Shukla, "NANOPRISM: A tool for evaluating granularity vs. reliability trade-offs in nano architectures," in *GLSVLSI*, 2004, pp. 109–112.

[6] K. N. Patel, I. L. Markov, and J. P. Hayes, "Evaluating circuit reliability under probabilistic gate-level fault models," in *IWLS*, 2003.

[7] M. R. Choudhury and K. Mohanram, "Reliability analysis of logic circuits," *TCAD*, vol. 28, no. 3, pp. 392–405, March 2009.

[8] S. Krishnaswamy, S. M. Plaza, I. L. Markov, and J. P. Hayes, "Signature-based SER analysis and design of logic circuits," *TCAD*, vol. 28, no. 1, pp. 74–86, Jan. 2009.

[9] Q. Zhou and K. Mohanram, "Gate sizing to radiation harden combinational logic," *TCAD*, vol. 25, no. 1, pp. 155–166, 2006.

[10] R. Garg, N. Jayakumar, S. P. Khatri, and G. Choi, "A design approach for radiation-hard digital electronics," in *DAC*, 2006, pp. 773–778.

[11] K. Keutzer, "Dagon: technology binding and local optimization by dag matching," in *DAC*, 1987, pp. 341–347.

[12] M. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering," *IEEE Design and Test*, vol. 16, no. 3, pp. 72–80, 1999.

[13] http://www.opencelllibrary.org, 2008.